C++ template

```
template <class T> //T is a type parameter
T max(T x, T y){
   if (x>y) return x; else return y;
}
```

The preprocessor instantiates the template for every function call statement in the program creating an ordinary function. **max(10,20)** creates a function obtained by replacing T with int in the template.

C++ template

```
template <class T>
void sort(T list[ ], int length){
    int i, j;
    T temp;
    for(i=0; j<=length-1; j++)
          for(j=i+1; j<=length-1; j++)
                if(list[i]>list[j]){
                      temp=list[i];
                      list[i]=list[j];
                      list[j]=temp;
                } //if
}
....
float B[100]={…};
sort(B,100);
char C[20]={…};
sort(C, 20);
```

Functions as Parameters

- Often it is needed to have a function receiving another function as a parameter. Some languages such as Ada do not allow parameters to be functions
- Pascal allows function parameters

```
procedure integrate(function f(x:real): real; L,U: real; var result: real);
  var
    temp:real;
    …
begin
  …
  temp:=f(L);
  …
end;
```

C/C++

- Allow pointers to functions as parameters
- The function prototype is provided to be able to do type checking

```
void f(int l, int (* g) (int, int)){
   int j, k, l;
   j=(*g)(k, l);
 …
}
```

Referencing environment for executing a passed function

In languages that allow nested subprograms such as Javascript there three choices:

- *Shallow binding*: the environment of the call statement of the passed function
- *Deep binding*: the environment of the definition of the passed subprogram
- *Ad hoc binding*: the environment of the call statement that passed the function as an actual parameter

Referencing environment for executing a passed function

```
void main(){ …f(k);… }
void f(g){… g(); …}
void k( ){ …}
```

How to determine referencing environment of K?

deep binding: K definition (static scope)

shallow binding: environment of **g()** statement inside f

ad hoc: environment of **f(k)** statement inside main

Example

```
//Javascript syntax
function sub1( ){ //environment of of sub2 [deep binding]
    var x;
    function sub2( ){
        alert(x); //Create a dialog box with value of x
    }
    function sub3( ){
        var x;
        x=3;
        sub4(sub2); // call statement that passed sub2 [ ad hoc]
    };
    function sub4(subx){
        var x;
        x=4;
        subx( ); // sub2 call statement [shallow binding]
    }
    x=1;
    sub3();
};
```

Output

- Shallow binding: 4
- Deep binding: 1
- Ad hoc: 3

Overloaded Subprograms

- An overloaded subprogram is a subprogram that has the same name as another subprogram in the same referencing envirnment
  - Every version must have a different protocol (prototype)
  - C does not allow overloading
  - C++, Java, Ada, and C# have overloaded constructors

Overloaded Subprograms

- C++, Java, and C# allow mixed mode operations and hence the return type cannot be used to distinguish an overloaded function

  //the following overloading is invalid

  int f(int a){…}

  double f(int b){…}

  int x;

  x+f(…);

  // f(…) might return int or double and the return type cannot be used to distinguish which f

- Ada does not allow mixed mode operations and hence the overloading above is valid

  x+f(…)

  // f(…) is considered int and the first f will be called

Overloading in Ruby

Ruby does not allow overloading. A new definition replaces the old one

```
def wrap(s)
      wrap(s, “()”)
end
def wrap(s, wrapper)
      wrapper[0,1] + s + wrapper[1,1]
   end
```

> **wrap(“x”)**

ArgumentError: wrong number of arguments (1 for 2)

>**wrap(“testing”,”[ ]”)**

“[testing]”

Overloading in Ruby

An alternative for failed overloading attempt:

```
def wrap(s, wrapper = “()”)
   wrapper[0,1] + s + wrapper[1,1]
end
```

> **wrap(“x”, “<>”)**

“<x>”

> **wrap(“x”)**

“(x)”

Closures

- Supported in many programming languages such as Racket, Ruby, and JavaScript
- A closure is a function/method that has the properties
  - Can be returned as a function value
  - Can be passed as a parameter between functions
  - Remembers the environment at time of creation and can always access this environment even though they may no longer be in scope

JavaScript example

Instead of

```
function Add5(X){return X+5;}
function Add10(X){return X+10;}
...
```

Using closures

```
function AdderFactory(y){
  return function(X){return X+Y;}
}
```

JavaScript example

```
function AdderFactory(y){
   return function(X){return X+Y;}
}

var Add5=AdderFactory(5); //closure 1
var Add10=AdderFactory(10); //closure 2
print(Add5(123)); //128, note closure1 still have access to y=5
print(Add10(123)); //133, note closure2 still have access to y=10
```

Nested functions and closures

```
function outside(x) {
  function inside(y) { return x + y;}
  return inside;
}
```

```
f = outside(3);
print(f(5)); // 8
print(ouside(3)(5)); // 8
```

JavaScript Example

- Create five elements labeled "link 0", "link 1", …, "link 4“.
- On clicking on "link 0" should alert 0, clicking on "link 1" should alert 1, ….

Wrong attempt

```
function addLinks(){
  for(var i=0; i<5; i++){
    link=document.createElement(…);
    link.innerHTML = "link "+ i;
    link.onclick = function(){alert(i);}
    document.body.appenChild(link);
  }
  window.onload=addLinks;
```

Create 5 links labeled correctly but clicking any of the 5 links displays 5

Correct implementation

```
function addLinks(){
    for(var i=0; i<5; i++){
        link=document.createElement(…);
        link.innerHTML="link “ + i;
        link.onclick=function(){
                return function(x){alert(x);} (i);
                // 5 closures x=1 in first closure, x=2 in second, …
                // “link 1” onclick executes closure where x =1
                // “link 2” onclick executes closure where x = 2
                …
        }
        document.body.appenChild(link);
        }
}
window.onload=addLinks;
```

Public and private using closure

```
var person = function(){
    var name="ali“; // local variable (private)
    //public
    return{ // returns 2 closures that have access to name
            getName: function () {return name;}
            setName: function (newName){ name = newName; }
          };//return
} (); //execute and store the value of the function (2 closures) in person

alert(person.name); //undefined
alert(person.getName()); // executes closure and prints ali
person.setName("jamal"); // executes the other closure
alert(person.getName()); // jamal
```

Two choices for closure implementation

- The closure creates a copy of all the variables that it needs when it is created. The copies of the vars come along for the ride as the closure gets passed around. Every closure has a separate copy of the environment
- The closure extends the lifetime of all the vars that it needs. It will not copy them, but retain a reference to them and the vars will not be eligible for garbage collection while the closure is around. Multiple closures have access to the same environment
- Racket follows the first choice
- Ruby follows the second choice

Closure implementation

```
function f( ){
   int x=5;
   return function( ){x++; print(x);}
}
c1 = f( );
c2 = f( );
```

In Ruby c1, c2 have access to the same x

In Racket c1, c2 have separate copies of x